Accessing data with MySQL

Voy a seguir principalmente la guía oficial de Spring:

<https://spring.io/guides/gs/accessing-data-mysql/>

Voy a comentar en diversos lugares, y agregaré otras fuentes.

# Opciones

Hay varias opciones para acceder a la base de datos. Acá vamos a usar MySQL y JPA.

# Crear el proyecto Spring Boot

Creo el proyecto con la extensión de VS Code para Initializr.

ctrl + shift + p

Java: Create Java project…

Spring Boot

Maven Project

2.6.6

Java

tup

simple

11

Jar

## Dependencias

Spring Web

Spring Data JPA

MySQL Driver

SpringBoot DevTools

# Crear la base de datos y el usuario

El script para crear la base de datos y el usuario está en simple.sql en este mismo directorio. Yo lo usé en phpMyAdmin de WampServer.

# pom.xml

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-web</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-devtools</artifactId>

            <version>2.6.7</version>

        </dependency>

        <dependency>

            <groupId>mysql</groupId>

            <artifactId>mysql-connector-java</artifactId>

            <scope>runtime</scope>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-test</artifactId>

            <scope>test</scope>

        </dependency>

# src\main\resources\application.properties

spring.datasource.url=jdbc:mysql://localhost:3306/simple

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=springuser

spring.datasource.password=ThePassword

spring.jpa.hibernate.ddl-auto=update

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

# src\main\java\tup\simple\SimpleApplication.java

package tup.simple;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SimpleApplication {

    public static void main(String[] args) {

        SpringApplication.run(SimpleApplication.class, args);

    }

}

# src\main\java\tup\simple\controllers\UserController.java

package tup.simple.controllers;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.ResponseBody;

import tup.simple.models.User;

import tup.simple.repositories.UserRepository;

@Controller // This means that this class is a Controller

@RequestMapping(path = "/user") // This means URL's start with /user (after Application path)

public class UserController {

    @Autowired // This means to get the bean called userRepository

    // Which is auto-generated by Spring, we will use it to handle the data

    private UserRepository userRepository;

    @PostMapping(path = "/add") // Map ONLY POST Requests

    public @ResponseBody String addNewUser(@RequestParam String name, @RequestParam String email) {

        // @ResponseBody means the returned String is the response, not a view name

        // @RequestParam means it is a parameter from the GET or POST request

        User n = new User();

        n.setName(name);

        n.setEmail(email);

        userRepository.save(n);

        return "Saved";

    }

    @GetMapping(path = "/all")

    public @ResponseBody Iterable<User> getAllUsers() {

        // This returns a JSON or XML with the users

        return userRepository.findAll();

    }

    @GetMapping(path = "")

    public @ResponseBody String hola() {

        return "Hola";

    }

}

# src\main\java\tup\simple\models\User.java

package tup.simple.models;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Entity // This tells Hibernate to make a table out of this class

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Integer id;

    private String name;

    private String email;

    public Integer getId() {

        return id;

    }

    public void setId(Integer id) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public String getEmail() {

        return email;

    }

    public void setEmail(String email) {

        this.email = email;

    }

}

# src\main\java\tup\simple\repositories\UserRepository.java

package tup.simple.repositories;

import org.springframework.data.repository.CrudRepository;

import tup.simple.models.User;

// This will be AUTO IMPLEMENTED by Spring into a Bean called userRepository

// CRUD refers Create, Read, Update, Delete

public interface UserRepository extends CrudRepository<User, Integer> {

}

# Tests

En este momento, si en la consola sale un mensaje acerca de tests que no se han podido ejecutar, ir a la carpeta:

src\test\java\tup\simple\SimpleApplicationTests.java

y comentar toda la clase. También se puede borrar, pero por las dudas la dejamos. Esto es para que no corra los test, que por ahora no nos interesan.

# Probar

Ya podemos probar con:

<http://localhost:8080/user/>

<http://localhost:8080/user>

y nos responderá con Hola. La barra final, después de user, no es necesaria.

# Agregar usuarios

En la consola de Windows, NO en el PowerShell de Visual Studio, ni en el browser:

curl localhost:8080/user/add -d "name=Eulogia" -d "email=eulogia@gmail.com"

curl localhost:8080/user/add -d "name=Filiberto " -d "email=filiberto@gmail.com"

curl localhost:8080/user/add -d "name=Lindor" -d "email=lindor@gmail.com"

curl localhost:8080/user/add -d "name=Nicasio" -d "email=nicasio@gmail.com"

La respuesta es "Saved" en cada caso.

También lo podríamos hacer en PostMan.

# Listar usuarios

<http://localhost:8080/user/all>

[{"id":1,"name":" Uno ","email":" uno @gmail.com "},{"id":2,"name":" Dos ","email":" dos @gmail.com "},{"id":3,"name":" Other ","email":" other @gmail.com "},{"id":4,"name":" Some ","email":" some @gmail.com "},{"id":5,"name":" First ","email":" first @gmail.com "},{"id":6,"name":" Primero ","email":" primero @gmail.com "},{"id":7,"name":" Segundo ","email":" segundo @gmail.com "},{"id":8,"name":" Tercero ","email":" tercero @gmail.com "},{"id":9,"name":" Cuarto ","email":" cuarto @gmail.com "},{"id":10,"name":"Eulogia","email":"eulogia@gmail.com"},{"id":11,"name":"Filiberto ","email":"filiberto@gmail.com"},{"id":12,"name":"Lindor","email":"lindor@gmail.com"},{"id":13,"name":"Nicasio","email":"nicasio@gmail.com"}]

Esto no es un objeto de JavaScript, sino una string de JSON correctamente formada, que representa un array

[]

de objetos

{}

cada uno de los cuales tiene dentro tres atributos

"nombre\_del\_atributo":valor\_del\_atributo

# Discusión

La clase principal es DemoApplication. Lo único que hace, desde nuestro punto de vista, es arrancar la aplicación, de modo que TomCat estará listening en el puerto 8080. Con la aplicación andando, la clase UserController se hace cargo de los requests.

Cuando nosotros escribimos en la consola de Windows:

curl localhost:8080/user/add -d "name=algún\_nombre" -d "email=algun@gmail.com"

estamos haciendo un request POST a nuestra aplicación. La URL termina con /user/add. La primera parte, o sea /user, la encontramos en la línea 15 de la clase UserController:

@*RequestMapping*(path = "/user") // URL's start with /user (after Application path)

El comentario explica lo que está pasando. La segunda parte, /add, la encontramos en la línea 21 de la misma clase:

    @*PostMapping*(path = "/add") // Map ONLY POST Requests

El comentario explica lo que está pasando. La URL que escribimos en la invocación a curl es mapeada al método addNewUser(). Este método toma del body request POST los dos parámetros que tiene como argumentos, crea un nuevo objeto de tipo User, puebla las propiedades usando los parámetros, e invoca al método save() del objeto userRepository, que es de tipo UserRepository. Si vamos a la interfaz UserRepository veremos que es muy corta. Declara que extiende la interfaz CrudRepository, pero no vemos que implemente nada. CrudRepository, a su vez, declara varios métodos pero no los implementa. La anotación @Autowired[[1]](#footnote-1), [[2]](#footnote-2) es la que hace el trabajo. Utiliza un patrón llamado inyección de dependencias, para evitar que escribamos código boiler plate, es decir cosas que son obvias, muy sabidas, repetitivas, que no requieren pensar y se pueden generar automáticamente.

En la documentación de la interfaz CrudRepository[[3]](#footnote-3) está la lista de los métodos disponibles, que se pueden usar sin necesidad de programar la implementación.

1. <https://www.baeldung.com/spring-autowire> [↑](#footnote-ref-1)
2. <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html> [↑](#footnote-ref-2)
3. <https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html> [↑](#footnote-ref-3)